Chapter 4

Results

In this chapter, we sum up the results of the research of this thesis. We do not present an exhaustive list of the results of each complementary publication of ours. Instead, we illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 3.

4.1 RQ1. To what extent can we generate program variants for WebAssembly?

As we describe in Section 3.2, our first research question aims to answer how to generate WebAssembly program variants. We pass each function of the corpora listed in Table 3.1 to CROW, and we collect how many variants CROW generates for each function. This section is organized as follows. First we present the general results for Metric 1 for each corpus. Second, we discuss the challenges and limitations attempting against the generation of program variants. Third, we enumerate and highlight properties of code that leverage more program variants. Finally, we illustrate the most common code transformations and answer RQ1.

General results

We summarize the results in Table 4.1. We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants.

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the diversified programs with their variants.

For example, the size change ratio between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for Rosetta , Libsodium, and QrCode. This limited impact on the binary size of the variants is good news because they are meant to save bandwidth when they become assets to distribute over the network.

CORPUS	#Functions	# Diversified	# NonDiversified	# Variants
Rosetta	303	**239**	64	1906
Libsodium	869	**85**	784	4272
QrCode	1849	**32**	1817	6369

Table 4.1: General diversification results. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the number of non-diversified functions and the cumulative number of variants.

Challenges for automatic diversification

We generate variants for functions in three corpora. However, we have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions, as it can be appreciated in Table 4.1. Our approach successfully diversified approx. 79 %, 9.78 % and 1.73 % of the original functions for Rosetta , Libsodium and QrCode respectively. On the other hand, we generated more variants for QrCode, 6369 program variants for 32 diversified functions.

Not surprisingly, setting up the timeout affects the capacity for diversification. A low timeout for exploration gives our approach more power to combine code replacements. We can appreciate this in the last column of the table, where for a lower number of diversified functions, we create, overall, more variants.

Moreover, we look at the cases that yield a few variants per function. There is no direct correlation between the number of identified codes for replacement and the number of unique variants. Therefore, we manually analyze programs that include many potential places for replacements, for which we generate few or no variants. We identify two main challenges for diversification.

1) Constant computation We have observed that our approach searches for a constant replacement for more than 45% of the blocks of each function while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because our tool is oblivious to a memory model.

2) Combination computation The overlap between code blocks, is a second factor that limits the number of unique variants. We can generate a high number of variants, but not all replacement combinations are necessarily unique.

Properties for large diversification

We manually analyzed the programs that yield more than 100 unique variants to study the critical properties of programs leveraging a high number of variants. This reveals one key reason that favors many unique variants: the programs include bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring [**?**] is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. Souper, the synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 4.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 4.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 4.3), using the intrinsics of the underlying computation model.

Listing 4.1: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 4.2: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

Listing 4.3: Diversification through overflow of integer operands.

```
i32.const 2
i32.mul
```

```
i32.const 2
i32.mul
i32.const -2147483647
i32.mul
```

4.2 Answer to RQ1.

We can provide diversification for more than 70% of the study cases. However, even when we cannot diversify some functions due to timeout, the overall number of created variants tripled the order of magnitude of the original functions count. This is the first realization of automated diversification for WebAssembly to the best of our knowledge.

4.3 RQ2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant to collect their execution traces and execution times. For each programs' population we compare Metric 2 and Metric 3 for each program/variant pair. This section is organized

as follows. First, we analyze the programs' populations by comparing the values of Metric 2 for each pair of program/variant. The pairwise comparison will hint at the results at the population level. We want to analyze not only the differences of a variant regarding its original program, but we also want to compare the variants against other variants. Second, we do the same pairwise strategy for Metric 3, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

Stack operation traces.

In Figure 4.1 we plot the distribution of all dt_dyn comparisons for all pairs of program/variant of each program's population generated un RQ1. All compared programs are statically different. Each vertical violin plot (in logarithmic scale) represents the dt_dyn values for a program of the Rosetta corpus for which we generated variants. For illustration, we exclude those programs with only one variant. The X-axis is sorted in descending order of the size of the program's population. We want to remark that the programs excluded from the plot all result from their comparisons in dt_dyn larger than zero. Thus, the unique generated variants present a different behavior in terms of stack operation traces.

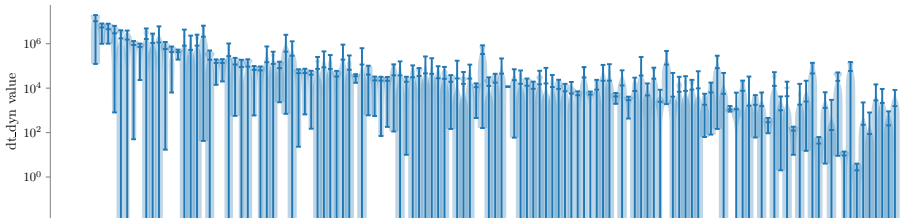

Figure 4.1: Pairwise of Metric 2 values in logarithmic scale. Each vertical plot represents a program and its variants. The plot only contains the programs for which we generate more than one variant, *i.e.,* more than one pair of programs comparisons.

We have observed that in the majority of the cases, the mean of the comparison values is huge in all cases. We analyze the length of the traces, and one reason behind such large values of dt_dyn is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant. We have observed no relation between how aggressive the transformation is and the length of the traces.

In some cases, even when the variants are statically different, two programs/variants result in a dt_dyn value oz zero, *i.e.,* they result in the same stack operation trace. We identified two main reasons behind this phenomenon. First,

the code transformation that generates the variant targets a non-executed or dead code. This result calls for future work on correct code debloating [?] . Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case. The four cases leave the same value in the stack operation trace.

```
(1) i32.lt_u      i32.lt_s      (3) i32.ne        i32.lt_u
(2) i32.le_s      i32.lt_u      (4) local.get 6   local.get 4
```

In Figure 4.2 we plot those programs for which at least one comparison (dt_dyn) is zero. The plot contains 82 vertical bars, one for each program. In the other cases, all dt_dyn values are non-zero. We can observe that even when some programs/-variants result in the same trace, there is no case for which all variants return the same stack operation trace with all bars above $Y = 0.2$. There is always at least one generated variant that is dynamically different from the original program.

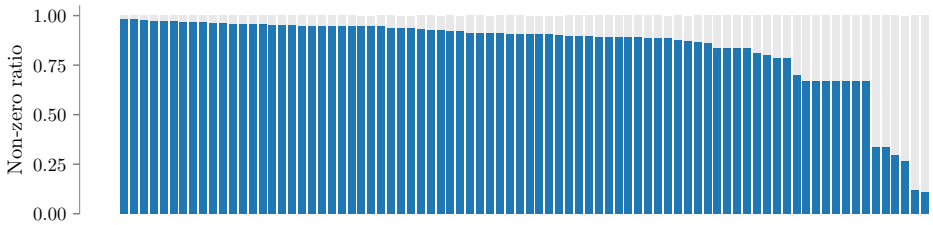

Figure 4.2: Ratio of Metric 2 with non-zero values for each program's population. Each vertical line represents the number of dt_dyn different from zero in a pairwise comparison of each program and its variants. The plot only contains the 82/239 with at least one pair comparison with zero value.

Execution times.

For each program's population, we compare the execution time distributions, Metric 3, of each pair of program/variant. Overall diversified programs, 169 out of 239 have at least one variant with a different execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for two programs' populations.

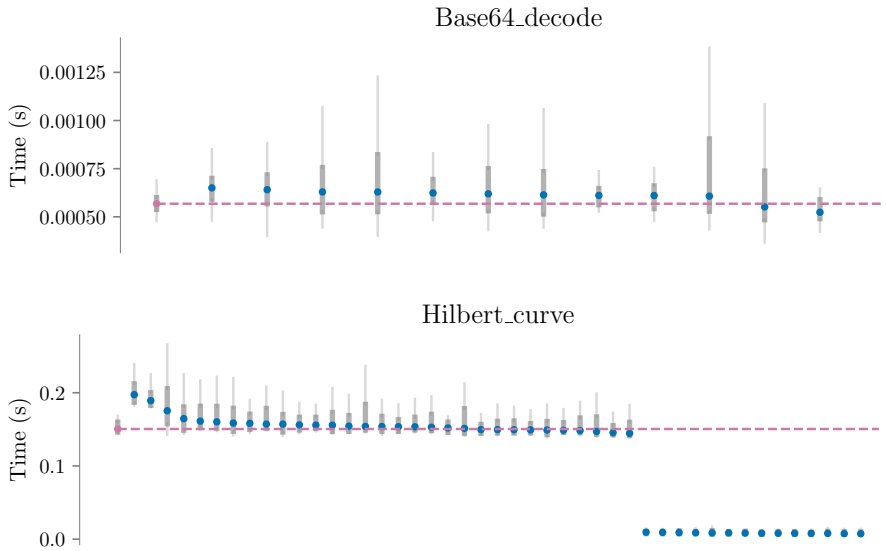

Figure 4.3: Execution time distributions for `Base64_decode` and `Hilbert_curve` program and their variants in top and bottom figures respectively. Baseline execution time mean is highlighted with the magenta horiontal line.

The plots in Figure 4.3 show the execution time distributions of programs `Base64_decode` and `Hilbert_curve` and their variants. We illustrate time diversification with these two programs because, for both, we generate unique variants with all types of transformations previously discussed in Section 4.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in magenta color: the distribution of 100 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For `Base64_decode`, the majority of variants are constantly slower than the reference programs (blue dot above the magenta line). For `Hilbert_curve`, many diversified variants are optimizations (blue dots below the magenta bar). The case of `Hilbert_curve` is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding the trade-off between taking all variants or only the ones providing the same or less execution time for the sake of less overhead.

4.4 Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which their execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. For example, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a more resilient execution.

4.5 RQ3. To what extent can the artificial variants be used to enforce security on Edge-Cloud platforms?

The third and last research question investigates the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for the Libsodium and the QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their function call traces and execution times. We analyze the trace differences at the Edge node and internet levels. Finally, we analyze the execution time distribution of the multivariant binaries.

Multivariant binary traces.

We execute the multivariant binaries of each program on the Fastly edge-cloud infrastructure. We execute each endpoint 100 times on each of the 64 edge nodes. All the executions of a given endpoint are performed with the same input. After each execution of an endpoint, we collect the sequence of invoked functions, i.e., the execution trace.

Figure 4.4 shows the ratio of unique traces exhibited by each program on each of the 64 separate edge nodes. The X corresponds to the edge nodes. The Y-axis

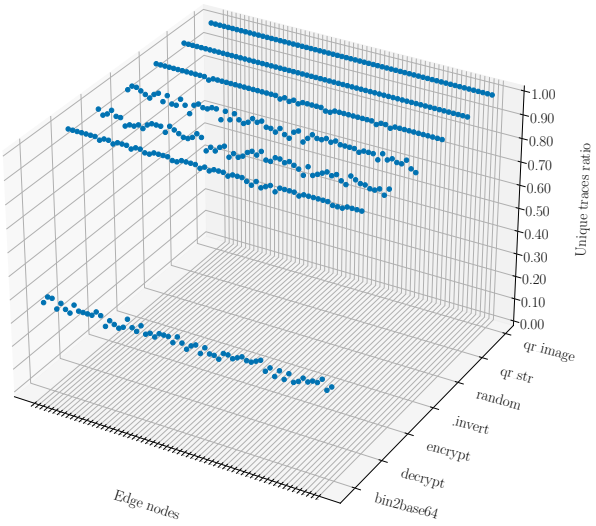

Figure 4.4: Ratio of unique execution traces for each endpoint on each edge node. The X axis illustrates the edge nodes. The Y axis annotates the name of the endpoint. In the plot, for a given (x,y) pair, there is blue point representing the Metric 4 value in a set of 100 collected execution traces.

gives the name of the endpoint. In the plot, for a given (x,y) pair, there is a blue point in the Z-axis representing Metric 4 over 100 execution traces.

For all edge nodes, the ratio of unique traces is above 0.38. In 6 out of 7 cases, we have observed that the ratio is remarkably high, above 0.9. These results show that we generate multivariant binaries that can randomize execution paths at runtime in the context of an edge node. The composition of the variants, associated with a significant number of function variants, significantly reduces the certainty about which computation is performed when running a specific input with a given input value.

Let's illustrate the phenomenon with the program `invert`. The program `invert` receives a vector of integers and returns its inversion. When the program is executed 100 times with the same input on the multivariant binary, we observe between 95 and 100 unique execution traces, depending on the edge node. We observe that the traces include only two invocations of the composition of variants for two different

functions, one at the start of the trace and one at the end. The remaining events in the trace are fixed each time the program is executed with the same input we provide in our experiments. Thus, the maximum number of possible unique traces is the multiplication of the number of variants for each involved function, in this case, $29 \times 96 = 2784$. The probability of observing the same trace is $1/2784$.

For multivariant binaries that embed only a few variants, like in the case of the `bin2base64` program, the ratio of unique traces per node is lower than for the other programs. With the input we pass to `bin2base64`, the execution trace includes 57 function calls. We have observed that this program selects among 41 variants of one diversified function. Thus, the probability of having the same execution trace twice is $1/41$.

Meanwhile, `qr_str` embeds thousands of variants, and the input we pass triggers the invocation of 3M functions, for which 210666 random choices are taken relying on 17 variants' populations. Consequently, the probability of observing the same trace twice is minimal. Indeed, all the executions of `qr_str` are unique, on each separate edge node.

We build the union of all the execution traces collected on all edge nodes for a given program. Then, we compute the normalized Shannon Entropy over this set for each endpoint (Metric 5). Our goal is to determine whether the diversity of execution traces we previously observed on individual nodes generalizes to the whole edge-cloud infrastructure. Depending on many factors, such as the random selection of variants during runtime, we might observe different traces on individual nodes, but the set of traces is the same on all nodes.

The second column of Table 4.2 gives the normalized Shannon Entropy value (Metric 5). Columns 3 and 4 give the median and the standard deviation for the length of the execution traces. Columns 5 and 6 give the number of diversified functions involved in the programs' execution (#Diversified) and the total number of invocations of these programs (#Runtime choices). These last two columns indicate to what extent the execution paths are randomized at runtime. In the cases of `invert` and `random`, both have the same number of taken random choices. However, the number of variants to choose in `random` is larger. Thus, the entropy is larger than `invert`.

Overall, the normalized Shannon Entropy (Metric 5) is above 42%. This is evidence that the multivariant binaries generated can exhibit a high degree of execution trace diversity while keeping the same functionality. The number of randomization points along the execution paths (#Runtime choices) is at the core of these high entropy values. For example, every execution of the `encrypt` endpoint triggers 4M random choices among the different function variants embedded in the multivariant binaries. Such a high degree of randomization is essential to generate diverse execution traces.

The `bin2base64` endpoint has the lowest level of diversity. This endpoint is the one that has the least variants, and its execution path can be randomized only at one function. The low level of unique traces observed on individual nodes is reflected at the system-wide scale with globally low entropy.

Endpoint	Entropy	Mean Trace Length	σ	#Diversified	#Runtime choices
libsodium					
encrypt	0.87	816	0	5	4M
decrypt	0.96	440	0	5	2M
random	0.98	15	5	2	12800
invert	0.87	7343	0	2	12800
bin2base64	0.42	57	0	1	6400
qrcode-rust					
qr_str	1.00	3045193	0	17	1348M
qr_image	1.00	3015450	0	15	1345M

Table 4.2: Execution trace diversity over the edge-cloud computing platform. The table is formed of 6 columns: the name of the program, the normalized Shannon Entropy value (Metric 5), the median size of the execution traces, the standard deviation for the trace lengths the number of executed dispatchers (#Diversified) and the number of total random choices taken during all the 6400 executions (#Runtime choices).

For both `qr_str` and `qr_image` the entropy value is 1.0. This means that all the traces that we observe for all the executions of these endpoints are different. In other words, someone who runs these services repeatedly with the same input cannot exactly know which code will be executed in the next execution. These very high entropy values are made possible by the millions of random choices that are made along the execution paths of these endpoints.

While there is a high degree of diversity among the traces exhibited by each endpoint, they all have the same length, except in the case of `random`. This means that the entropy is a direct consequence of randomly executing different function variants. In the case of `random`, it naturally has a non-deterministic behavior. Meanwhile, we observe several calls to variants composition during the execution of the multivariant binary, which indicates that we amplify the natural diversity of traces exhibited by `random`. For each endpoint, we managed to trigger all dispatchers during its execution.

Timing side-channels at program level.

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4.5, each subplot represents the distribution for a single program, with the colors blue and green representing the original and multivariant binary,

respectively. These plots reveal that the execution times are spread over a more extensive range of values than the original binary. This is evidence that execution time is less predictable for multivariant binaries than original ones.

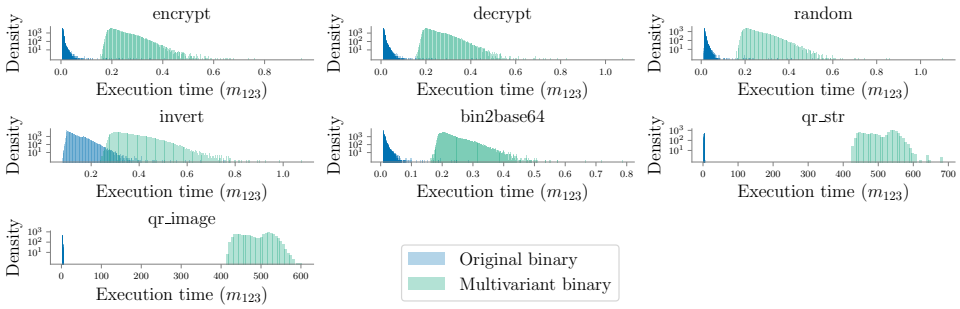

Figure 4.5: Execution time distributions. Each subplot represents the distribution for a single program, blue for the original program and green for the multivariant binary. The X axis shows the execution time in milliseconds and the Y axis shows the density distribution in logarithmic scale.

Recall that the choice of function variant is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

4.6 Answer to RQ3.

Repeated executions of a multivariant binary with the same input on an individual edge node exhibit diverse execution traces. Our approach of combining function variants in a single function successfully triggers diverse execution paths at runtime on individual edge nodes. At the internet-scale of the Edge platform, the multivariant binaries exhibit a massive diversity of execution traces while still providing the original functionality. An attacker can't predict which is taken for a given query. The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from execution times gathered from the multivariant binary. Therefore, we contribute to mitigate potential attacks based on predictable execution times.

4.7 Conclusions

This work proposes and evaluates our approach to generate WebAssembly program variants artificially. Our approach generates different, both statically and dynamically, variants for up to 70% of the programs in our three corpora. While generating statically different programs is still important, we highlighted the importance of complementing static and dynamic studies for programs diversification. We generate variants that exhibit a significant diversity of execution times that we can use to compose time-unpredictable multivariant binaries. Our approach of combining function variants in a single function as multivariants successfully triggers diverse execution paths and execution times at runtime. Therefore, making it virtually impossible for an attacker to predict which code is executed for a given query. Finally, we demonstrate the feasibility of automatically generating WebAssembly program variants.